ESD-TR-77-259

MTR-3294
Vol. I

DESIGN AND ABSTRACT SPECIFICATION OF A MULTICS SECURITY KERNEL

NOVEMBER 1977

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Bedford, Massachusetts

Project No. 522N
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. AF19628-77-C-0001

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

William R. Price
WILLIAM R. PRICE, Capt, USAF
Techniques Engineering Division

Roger R. Schell
ROGER R. SCHELL, Lt Col, USAF
ADP System Security Program Manager

FOR THE COMMANDER

Stanley P. Dereska
STANLEY P. DERESKA, Colonel, USAF
Deputy Director, Computer Systems Engineering
Deputy for Command & Management Systems

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-77-259	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DESIGN AND ABSTRACT SPECIFICATION OF A MULTICS SECURITY KERNEL		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER MTR-3294, Vol. I
7. AUTHOR(s) W. L. Schiller		8. CONTRACT OR GRANT NUMBER(s) AF19628-77-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation Box 208 Bedford, MA 01730		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 522N
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Command and Management Systems Electronic Systems Division, AFSC Hanscom Air Force Base, Bedford, MA 01731		12. REPORT DATE NOVEMBER 1977 13. NUMBER OF PAGES 58
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

COMPUTER SECURITY
FORMAL SOFTWARE SPECIFICATION
MULTICS
SECURITY KERNEL

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

On the basis of the recommendations of the ESD Computer Security Technology Panel (1972), MITRE developed techniques for the design, implementation, and formal mathematical verification of a security kernel: a hardware and software mechanism to control access to information within a computer system. This three-volume report describes the design of a security kernel for the Honeywell

20. ABSTRACT (concluded)

Information Systems Multics computer system. This first volume gives a methodology and design overview. The primary subsystems are defined, and the reasoning behind the design decisions is given. The correspondence of the design to a mathematical model is discussed, giving a preview to the formal verification. The second volume gives a formal top-level specification of the primary subsystems. The third volume deals with the secondary subsystems, including the issues of initialization and reconfiguration.

ACKNOWLEDGMENTS

This report has been prepared by The MITRE Corporation under Project 522N. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

This report describes a design that has been evolving since September 1974. A number of individuals have contributed, including S. R. Ames, Jr., K. J. Biba, E. L. Burke, M. Gasser, S. B. Lipner, P. T. Withington, and J. P. L. Woodward of The MITRE Corporation; and Lt. Col. R. R. Schell, Capt. W. R. Price, and Capt. P. A. Karger of the U. S. Air Force. The design has also been influenced by discussions with personnel from other Air Force contractors and subcontractors: Honeywell Information Systems, the Computer Systems Research group of M.I.T's Laboratory for Computer Science, and Stanford Research Institute's Computer Science group.

TABLE OF CONTENTS

TABLE OF CONTENTS (conc'd)

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION I

INTRODUCTION

For the past several years MITRE has been pursuing an Air Force Electronic Systems Division (ESD) sponsored program concerned with the development and demonstration of computer security technology. The basis of this work is a series of recommendations made by the ESD Computer Security Technology Panel.[1] The panel suggested that computer security could be achieved by developing a security kernel, a hardware and software mechanism that controls access to information stored in a computer system. The panel also suggested a formal mathematical approach for verifying the correctness of a kernel.[2] The outgrowth of this latter suggestion has been the development of a design methodology for highly reliable software.

In this three-volume report, we describe the design of a security kernel for the Honeywell Multics System. The design methodology is reviewed in Section II of this volume. In Section III, we give an overview of the kernel design, concentrating on design alternatives and the correspondence of the design to a mathematical security model that is part of the design methodology. This first volume concludes with an informal description of the functions provided by the three primary kernel subsystems. (The primary kernel subsystems are those subsystems that are important to each Multics process.)

In the second volume, a formal, top-level specification of the primary kernel subsystems is presented. The final volume gives a description and specification of the secondary kernel subsystems. In the remainder of this introductory section, we discuss the context of the kernel design.

DESIGN CONTEXT

The Honeywell Series 60/Level 68 hardware and its Multics operating system were chosen as the base for a prototype, large-scale, secure computer system for two principal reasons. First, a study completed in mid-1974 determined that the Multics hardware was the off-the-shelf large computer best suited to support a security kernel.[3]

Second, the Multics software was designed with security in mind and possesses sound security design. The belief was that a

secure Multics system can be constructed without significant change to the user interface. Furthermore, the current Multics implementation is modular and coded in a high level language (PL/I). These characteristics make the Multics system more understandable and more amenable to redesign.[4] Thus, using Multics as the hardware-software base should minimize the cost of developing a large-scale, kernel-based, secure system.

Initial steps toward developing a secure system based on Multics were taken in conjunction with development of a Multics operating system for use in a two-level (Secret and Top Secret) environment at the Air Force Data Services Center. This system's design is aimed at providing security controls based on the military access rules, but it does not claim to eliminate completely the prospect of hostile penetration. The resulting Multics enhancements are called AIM, for Access Isolation Mechanism.[5] These enhancements demonstrate the impact on the Multics user interface of using military security policy to control access to information stored in Multics. The user interface of a kernel-based Multics will be equivalent to the AIM interface, because both systems will enforce the same security policy. The AIM interface is essentially compatible with the standard (i.e., non-AIM) Multics interface. AIM also demonstrates the inherent cost of security--all checks required by the military policy are made.

Since AIM is not implemented with a security kernel, its controls cannot be proven effective. Thus, the next step is the development of a security kernel so that a secure Multics can be used in an open (i.e., uncleared users), multilevel environment, where hostile penetration attempts are possible.

SECTION II

DESIGN METHODOLOGY

A key issue in computer security is the effectiveness of a system's access controls. Contemporary systems are notoriously easy to penetrate, and attempts to fix them have failed.[6, 7, 8] Recognizing the inadequacy of current techniques, the ESD computer security technology panel recommended that the developers of a secure system "start with a statement of an ideal system, a model, and to refine and move the statement through various levels of design into the mechanisms that implement the model system" [1]. In this section we will review the methodology used in designing the Multics security kernel. The heart of this methodology is a series of design stages or representations. Before these representations can be described, some basic concepts must be defined.

BASIC CONCEPTS

The basic component of our ideal system is a reference monitor--an abstract mechanism that controls accesses of subjects (active system elements) to objects (information repositories) (see Figure 1). Subjects are users or processes, and objects include programs, data files, and peripheral devices. The hardware-software mechanism that implements the reference monitor abstraction is called a security kernel. When the computer hardware is predetermined, the software that must be designed to implement the reference monitor abstraction is frequently referred to as the security kernel for that computer.

An implementation of the reference monitor abstraction permits or prevents access by subjects to objects, making its decisions on the basis of subject identity, object identity, and the security parameters of the subject and object. The implementation both mechanizes the access rules of the military security system and assures that they are enforced within the computer. To provide the basis for multilevel secure computer systems, a reference monitor implementation must: 1) mediate <u>every</u> access by a subject to an object; 2) be <u>protected</u> from unauthorized modification; and 3) <u>correctly</u> perform its function.

Creating a design that complies with the first two requirements is relatively easy. A kernel mediates every access by creating and controlling an environment in which all non-kernel software is constrained to operate. The kernel can be thought of as creating an abstract or virtual machine. If a security policy is correctly built

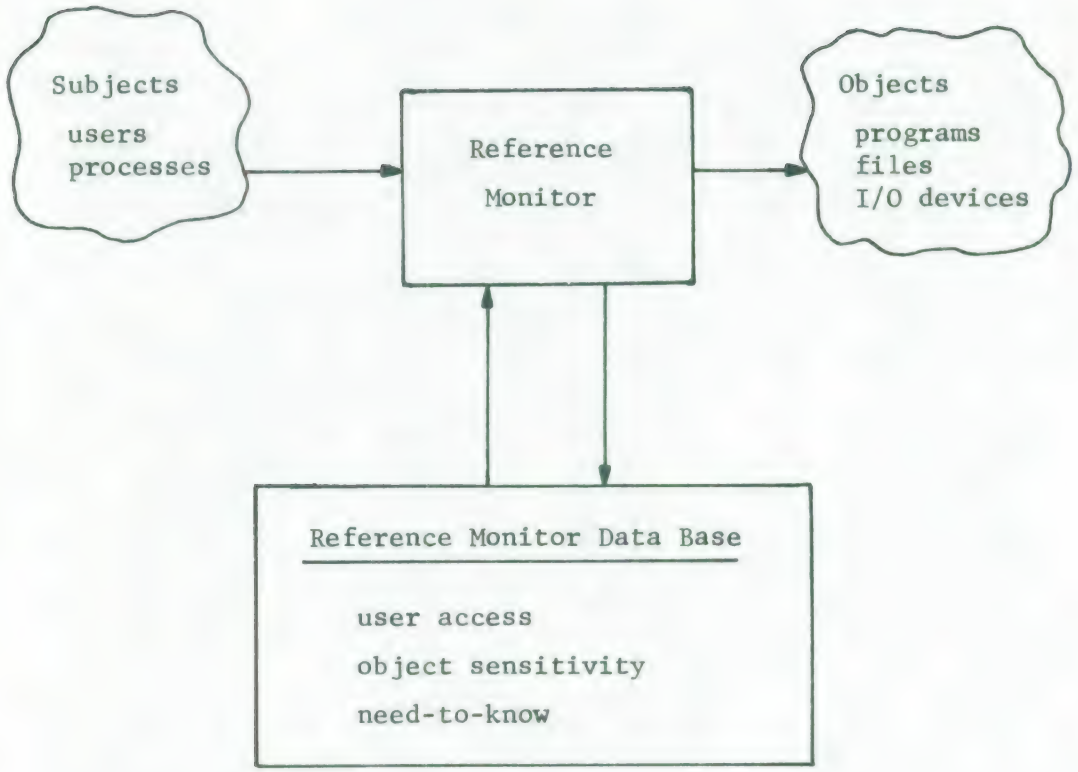


Figure 1. Reference Monitor

into the abstract machine, then programs running on it will not be able to perform operations that violate the policy. In practice, the abstract machine created by a security kernel will include all of the unprivileged machine instructions of the base hardware, constrained by a hardware supported memory protection mechanism, and functions implemented by kernel software. (The base hardware is the hardware on which the security kernel software executes.)

Kernel protection is achieved by isolating the security kernel software in one or more hardware supported protection domains, in the same manner that contemporary operating systems use protection domains to protect themselves from applications software.

Proving that a kernel correctly performs its function, the third requirement, is not an easy task. It is this requirement that motivates the need for a formal design and verification methodology. As mentioned above, our methodology employs a series of design representations.

DESIGN REPRESENTATIONS

The process of proving that the behavior of a kernel-based system is consistent with its security requirements is called security kernel verification. In our case, the requirements are defined by DoD security policy. Since there is a large conceptual distance between DoD regulations and a secure machine with a kernel loaded and running, intermediate representations have been introduced to facilitate the verification process.[9]

One intermediate representation is a mathematical model that formally defines the security requirements. A single model can be used in the verification of any security kernel that must enforce the security policy embodied by the model. The other intermediate representations are specific to a particular kernel design. These representations are: 1) a formal specification of the input/output behavior of the abstract machine created by the security kernel; and 2) the algorithmic implementation of the kernel--programming language source listings. The verification methodology calls for the establishment of a correspondence between each pair of representations as shown in Figure 2.

The Mathematical Model

The mathematical model of security defines the operation of an abstract, secure computer system embodying the Department of Defense (DoD) security policy.[10] The basic element of the model

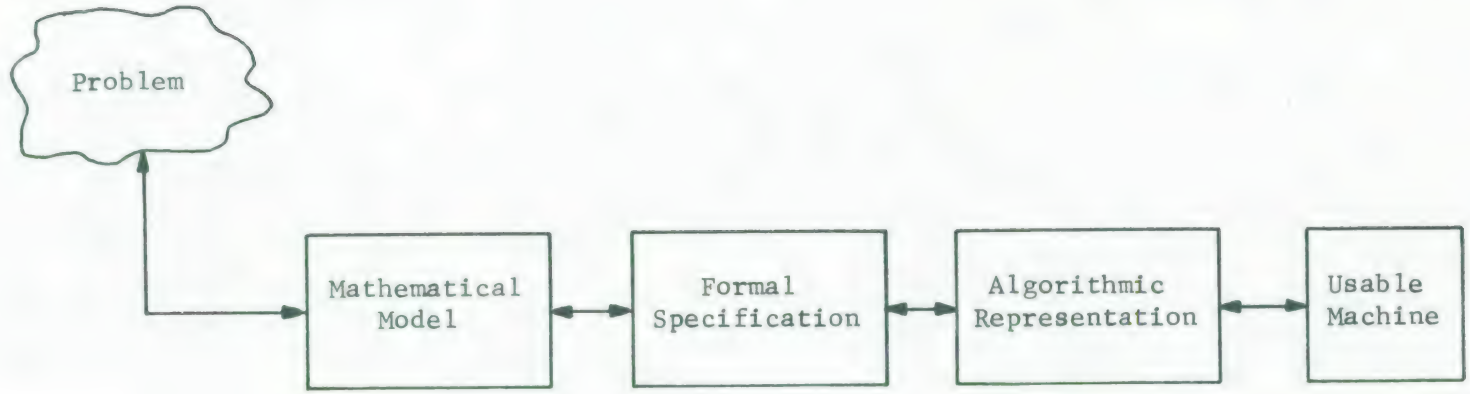


Figure 2. Verification Chain

are subjects, objects, their access levels, an access matrix, access attributes, and axioms. Subjects are the active accessors and objects the protected information receptacles. Every subject and object has an access level, which is a combination of a security level and integrity level. Security and integrity levels are, in turn, composed of a classification (usually from "unclassified" to "top secret") and a category set. Typical categories in DoD applications are "nuclear" and "NATO".

Security and integrity levels have a natural partial ordering, that is, an ordering in which some pairs of elements are not comparable. One security level is less than or equal to another security level if its classification is less than or equal to the other classification and its category set is included in the other category set. Similarly, integrity levels and access levels are partially ordered. Security levels are employed for protecting information from unauthorized observation, and integrity levels for protection from unauthorized modification. A discussion of integrity issues in secure systems, and the duality between security and integrity is given in [11].

The access matrix is used to implement "need-to-know" security. Its elements are accessed by subject identifier and object identifier, and each element indicates in what mode, if any, the specified subject may access the specified object.[12]

An access attribute describes whether information can be taken away from an object (observation), or dispatched to an object (modification). The correspondence between the actual access attributes used in the Multics kernel design and observation and modification is:

read = observation with no modification

write = both modification and observation

alter = modification with no observation

The axioms of the model define the information flow allowed by the security policy, and two additional properties required for a secure, consistent system state. Non-discretionary security (i.e., based on DoD security levels) is defined by the simple security condition and the *-property axioms:

Simple Security Condition

If a subject has observe access to an object, the access level of the object is less than or equal to that of the subject.

*-Property

If a subject has observe access to one object and modify to a second, the access level of the first (observe) is less than or equal to that of the second (modify).

The simple security condition follows directly from DoD security policy. In a computer system, the simple security condition must be supplemented by the *-property to prevent programs from accidentally or maliciously (i.e., a "Trojan Horse" [13]) copying information from a high access level object to a low access level object. To summarize the non-discretionary policy: a subject can only observe lower level objects, can only modify higher level objects, and can only observe and modify objects that have an access level equal to the subject's access level.

Within the constraints of non-discretionary security, individuals can use their own judgment in disclosing information to others ("need-to-know"). This policy is embodied in the discretionary security property:

Discretionary Security Property

If subject s has x access to object o, attribute x is recorded in element (s,o) of the access matrix.

The three axioms above define the security policy. Two additional axioms are necessary to define a consistent, secure system state. These axioms refer to "active" objects. The intent is that when an object becomes inactive, its contents are no longer available. The axioms are:

Tranquility Principle

The access levels of active objects do not change during normal operation.

Activity Principle

Accessible objects are active.

The model allows objects to have a tree-structure, that is, except for the root object, each object has a parent (see Figure 3). This structure corresponds (by design) to the hierarchical directory organization of the Multics virtual memory. A specific design can include some objects that are hierarchically organized and others that are not. For a reason that will be explained in the discussion

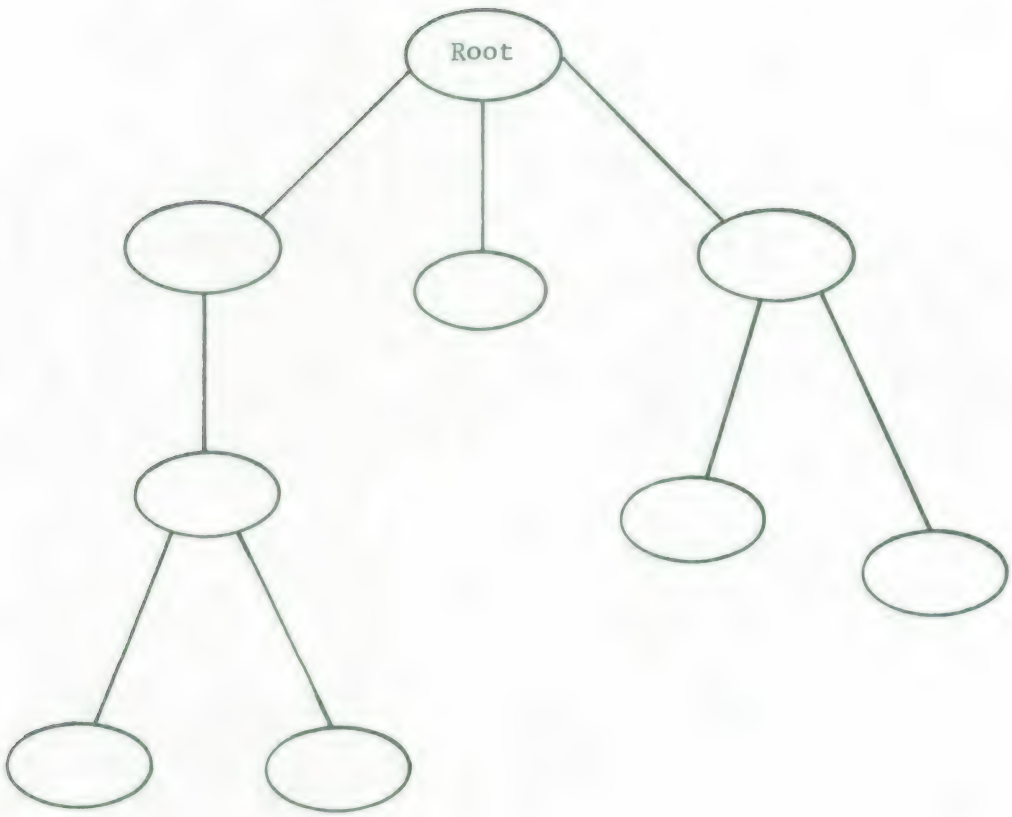


Figure 3. An Object Hierarchy

of the get access model rule, the access levels of hierarchically structured objects must be non-decreasing as one moves away from the root to the leaves. The existence of a hierarchy object is recorded in its parent--this information is at the level of the parent. The existence of a non-hierarchy object can be handled in one of two ways: 1) the existence of an object is information at the level of the object; or 2) knowledge of the object is controlled by a second access level that is distinct from the level of the object, a "visibility" access level. In the Multics kernel design, only the first approach is used.

Some Implications

Now that the model has been informally defined, we can discuss a few of its characteristics. First, there is a fundamental difference between non-discretionary and discretionary security. The nature of non-discretionary security allows global behavioral properties about the flow of information in the system with respect to access levels to be proved, but the same cannot be done for discretionary security. Specifically, a Trojan Horse can subvert a user's intentions for discretionary security, but it cannot circumvent the non-discretionary policy (except with "time" channels, see below).

The model is based on an asynchronous view of computation. Thus, it is possible for programs executing outside of a security kernel to influence the response time that other programs see, and to use this ability to modulate response time to send "Morse code".[14] Cooperating Trojan Horses at different levels can use a time channel to circumvent non-discretionary security. The kernel can reduce the bandwidth of time channels by adding noise, but closing them completely will significantly reduce the utility of a resource sharing computer system.[15]

The non-discretionary security axioms are applied not only to basic user information (file contents), but also to information about information--state variables created by kernel implementation mechanisms. Examples of state variables are the name, status, and disk address of a file. If the value of a state variable is a function of system-wide activity, its access level must be "system high." (System high is the maximum access level in the system. With respect to non-discretionary security, a system high subject may observe all information in a system, and any subject may modify a system high object.) Experience with an earlier kernel design has shown that secure access to state variables associated with resource usage (especially storage resources) can be most easily controlled by a complete virtualization on a per-user or per-access

level basis of all physical resources employed by the kernel in creating the user environment.[16] This approach requires the kernel to control resource usage with quota mechanisms.

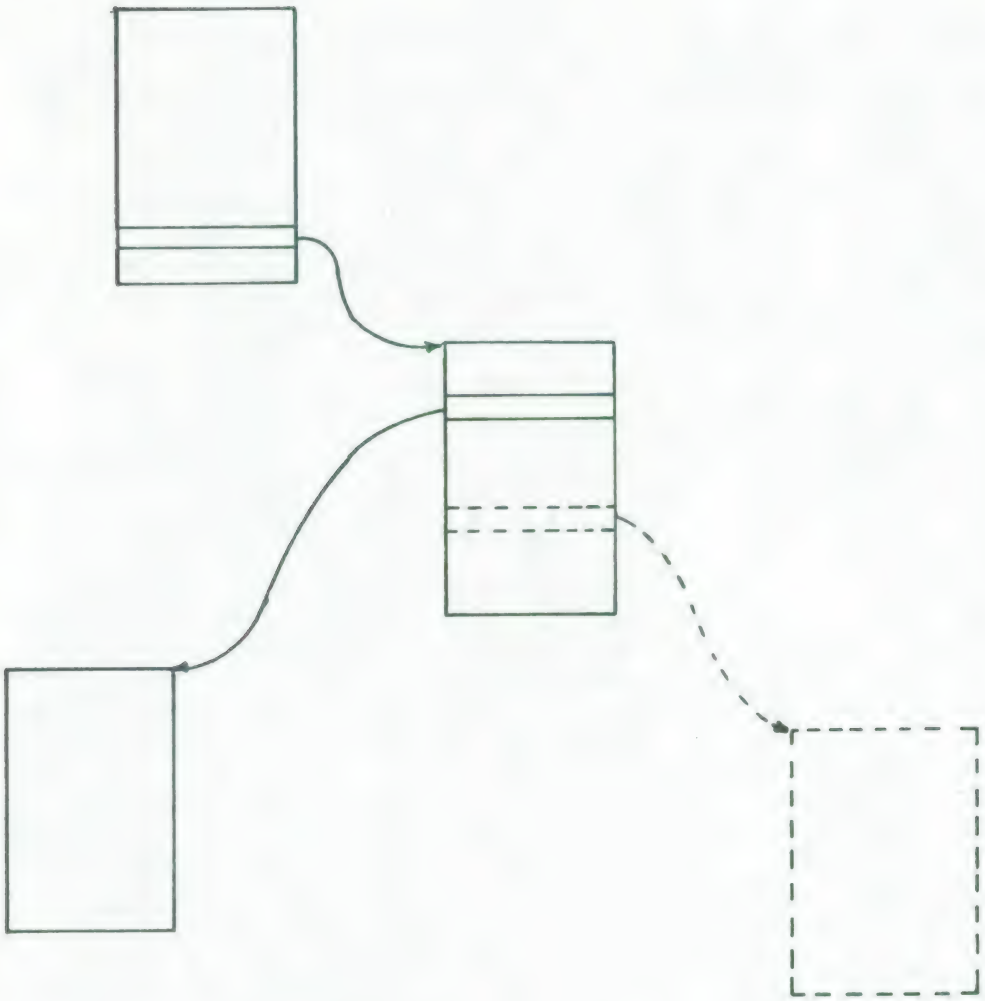
Model Rules

To demonstrate how the model works, a set of rules that describe the primitive operations of a secure system can be developed. If the rules are properly chosen, then each function provided by a specific security kernel design will correspond to a rule or a composition of rules. In [10], the following rules are suggested:

1. create object
2. delete object
3. give access
4. rescind access
5. get access
6. release access

As the rules and their security constraints are reviewed, it is important to remember that the rules simply demonstrate how the model works--they are an ideal implementation of the axioms. The formal verification of a security kernel involves analyzing the information flows that it permits with respect to the previously described axioms; model rules are not specifically used.

The security constraints on the create and delete object rules depend on whether or not the object is in a hierarchy. For hierarchy objects, create is interpreted as observation and modification of the parent object (see Figure 4). The parent must be modified because the existence of the new object is recorded in it. The parent must also be observed, because it is usually not practical to hide the occurrence of certain error conditions that are associated with the state of the parent. In particular, a create request must be refused if there is no free space in the parent object in which to record the existence of a new object, a situation that can occur in any system where objects have a finite size. Thus, to create a new object in a hierarchy a subject must have a level equal to the level of the intended parent.



Creating a new object in a hierarchy involves observation and modification of the intended parent object

Figure 4. Object Creation

A subject can only create a non-hierarchy object with an access level equal to or higher than its level because the *-property does not permit modification of lower level information.

The security constraints on object deletion are similar, except that a delete object rule can be provided that does not indicate the occurrence of error conditions (e.g., object not found). Thus, deletion can be modification without observation, and non-hierarchy objects at equal and higher levels as well as hierarchy objects with an equal or higher level parent can be deleted.

The give access and rescind access rules change the discretionary access permissions in the access matrix. At a minimum, these rules modify the access matrix. For the Multics design, only hierarchy objects have an explicit access matrix. The matrix is implicit for non-hierarchy objects--all subjects always have all discretionary access permissions to all non-hierarchy objects. Also, for the Multics design we associate the access matrix column for an object with that object's parent, and interpret the give access and rescind access rules as observation and modification of the parent object. Thus, a subject can give access and rescind access to an object if the subject is at the same level as the object's parent, contingent on the subject's discretionary access permission to the parent.

The get access rule moves an object into the set of objects that a subject can access in one or more specific access modes. The security requirements are the simple security condition, the *-property, and the discretionary security property. In addition, the get access of a hierarchy object requires observation of its parent object, because the objects existence, and the subject's discretionary access rights to it, are being determined. Thus, if an object has a level less than its parent's level, it can never be modified, because a subject cannot have an access level greater than or equal to the parent's (for observation) and less than or equal to the object's (for modification), as required by the simple security condition and *-property. Accordingly, access levels in a hierarchy must be non-decreasing as one moves from the root to the leaves. There are no security requirements on the release access rule.

The model permits a subject to access objects in its current access set in accordance with the axioms. Thus, observe object and modify object are implicit rules.

Specification

The next representation in the validation chain is a formal specification of the input/output behavior of the abstract machine

created by the kernel. For this representation we use a technique originally proposed by Parnas [17] and extended by Price [18].

A "Parnas" specification of an abstract machine or software module consists of two distinct types of functions: O-functions and V-functions. O-functions (operate) are functions that cause the state of the system to change. V-functions (value) return the values of state variables. The effect section of each O-function consists of specification statements. These statements denote that upon completion of the function, certain predicates will be true. The ordering of the specification statements is not significant and predicates can be conditional.

Parnas's intention for specification is to give an external view of functions. All of the information needed to correctly use functions and to implement them must be given, and nothing more. Also, specifications must be sufficiently formal so that their completeness, consistency, and other desirable properties (in our case, security correctness) can be determined.

For a reason that will be given shortly, the specification used at this stage is usually called the "top-level specification". Using a technique developed by Millen [19], a top level kernel specification can be systematically analyzed to determine if it complies with the model axioms. For the technique to work, it is essential that the specification include only the minimum information necessary to describe the input/output behavior of kernel operations. The mechanisms that support the operations must not be identified.

Algorithmic Representation

Given a kernel and its top-level specification, the next logical step is to implement the software portion of the kernel design on an appropriate hardware base. The corresponding step in the kernel verification process is the proof that the kernel implementation is correct with respect to its specification.† A methodology derived from a technique developed by a group at the Stanford Research Institute can be used for this purpose.[21]

The SRI technique employs the "standard" proof-of-correctness method (i.e., Floyd's inductive assertion method [22]). The problem of combinatorial explosion, which prevents the proof of programs much larger than a few hundred statements, is avoided by partitioning

†This requirement for correctness with respect to a specification has also been studied as part of a project to implement a secure virtual machine monitor for the PDP-11/45.[20]

the proof of a large program into the independent proof of many small programs. The partitioning is accomplished by decomposing the system design into a formally specified hierarchy of abstract machines, which represent levels of abstraction.[23]

Each level is specified in the same manner as the abstract machine created by the kernel as a whole is specified. To distinguish between the two sets of specifications, the former are called lower level specifications, and the latter is called the top-level specification. The specification of level n defines the input/output assertions for the proof of the abstract program that implements level n, running on level n-1.

Usable Machine

The final representations in the verification chain is a usable, secure system with a security kernel loaded and running. This representation is obtained by compiling the programs coded in developing the algorithmic representation. The verification requirement is a correct compilation. This requirement can be satisfied in at least two ways: 1) by proving the correctness of the compiler; or 2) by proving that the programs of interest have been correctly compiled. The complexity of contemporary compilers suggests that in the near term alternative 2) is more promising.

TRUSTED SUBJECTS

A secure computer system is not a closed system. A mechanism is required to associate the computer system elements correctly with their counterparts in the people/paper world. This function is performed by "trusted subjects": the active system entities that perform the security-related binding of computer system elements to the external environment.

Since the security correctness of a system depends on the proper operation of its trusted subjects, it is appropriate to verify the correctness of trusted subject software. Unlike security kernel verification, however, it is not particularly useful to verify trusted subjects with respect to a model. Rather, trusted subjects can be verified on a case by case basis. Trusted subjects provide functions to people to guide them in making the correct bindings. The computer system cannot internally determine the correct bindings or the correctness of the guidance.

Trusted subject software is usually controlled by a System Security Officer (SSO), an individual responsible for maintaining system security. In some cases trusted subject software can be controlled by any cleared user, where the user's clearance is commensurate with the binding to be performed. Trusted subject software can never be made available to uncleared users or unverified software, because neither can be trusted to define the correct bindings.

The security of a system depends on the correct operation of the software that implements the kernel and trusted subjects. Rather than introduce a new term, we refer to this software as kernel software, even though it includes trusted subject software. System security is independent of what non-kernel software does or does not attempt to do.

AUDIT AND SURVEILLANCE

For computer systems processing classified information, military security policy requires that an audit log be maintained as a history of the use of the system.[24] At a minimum, a system must be capable of recording in its audit:

1. each login,

2. each creation of a new classified file,

3. each access to a classified file and its nature, and

4. each production of accountable classified output.

As discussed by Engelman [25], audit and surveillance techniques can complement access controls in a kernel-based system. Audit and surveillance can be used to detect attempts by Trojan Horses to: 1) misuse discretionary security mechanisms, 2) employ time channels, and 3) probe for hardware failures.

SUMMARY

We have identified the basic concepts of an approach for providing secure computer systems, outlined a methodology for using these concepts to develop specific systems, and described the application of DoD security policy to a computer system environment. A key feature of the methodology is the mathematical verification that addresses the requirement for effective access controls.

SECTION III

DESIGN OVERVIEW

In this section we begin to describe the Multics security kernel. We start with a discussion of basic design principles, then review the structure of the current Multics system, and conclude with a description of the kernel subsystems.

DESIGN PRINCIPLES

Our goal is to design a Multics security kernel that enforces the policy described by the security model, and is compatible with the current Multics at a minimum cost in system efficiency. Since kernel verification is facilitated by a relatively small and well-structured kernel, it is desirable to make the abstract machine created by the kernel the minimum machine sufficient for the required function.

The goal of compatibility is secondary to security. Aspects of the current Multics that are fundamentally insecure will require modification and the possible introduction of incompatibilities.

Success in achieving the third goal, efficiency, is the most difficult to judge at this stage of the design. Since a kernel implementation, which can be measured, does not exist, the best we can do is to use our knowledge of the current Multics system structure to judge how the kernel design will affect efficiency. In some cases there will be a clear trade-off between a small, simple kernel and system efficiency. The efficiency cost of the kernel approach to security (or at least the cost of a particular Multics kernel design) can be measured by comparing a kernel-based Multics to an AIM Multics, since AIM already has all of the security checks required by military policy.

There are two general approaches for designing a Multics kernel: 1) synthesize a kernel by using the security model and the definition of the current Multics user interface as a starting point for a top-down design; and 2) analyze the structure of the existing Multics to determine how it can be transformed into a kernel-based system. Elements of both approaches have been employed in the evolution of the kernel design to its present state, and we will use both points of view in describing the design.†

---

†Work by Honeywell Information Systems on a Multics kernel design can be viewed as further refinement and evolution of the design presented here.

THE CURRENT MULTICS

To set the context for the kernel design, we will briefly review the major features of the current Multics system. A more complete description of the system can be found in [26, 27].

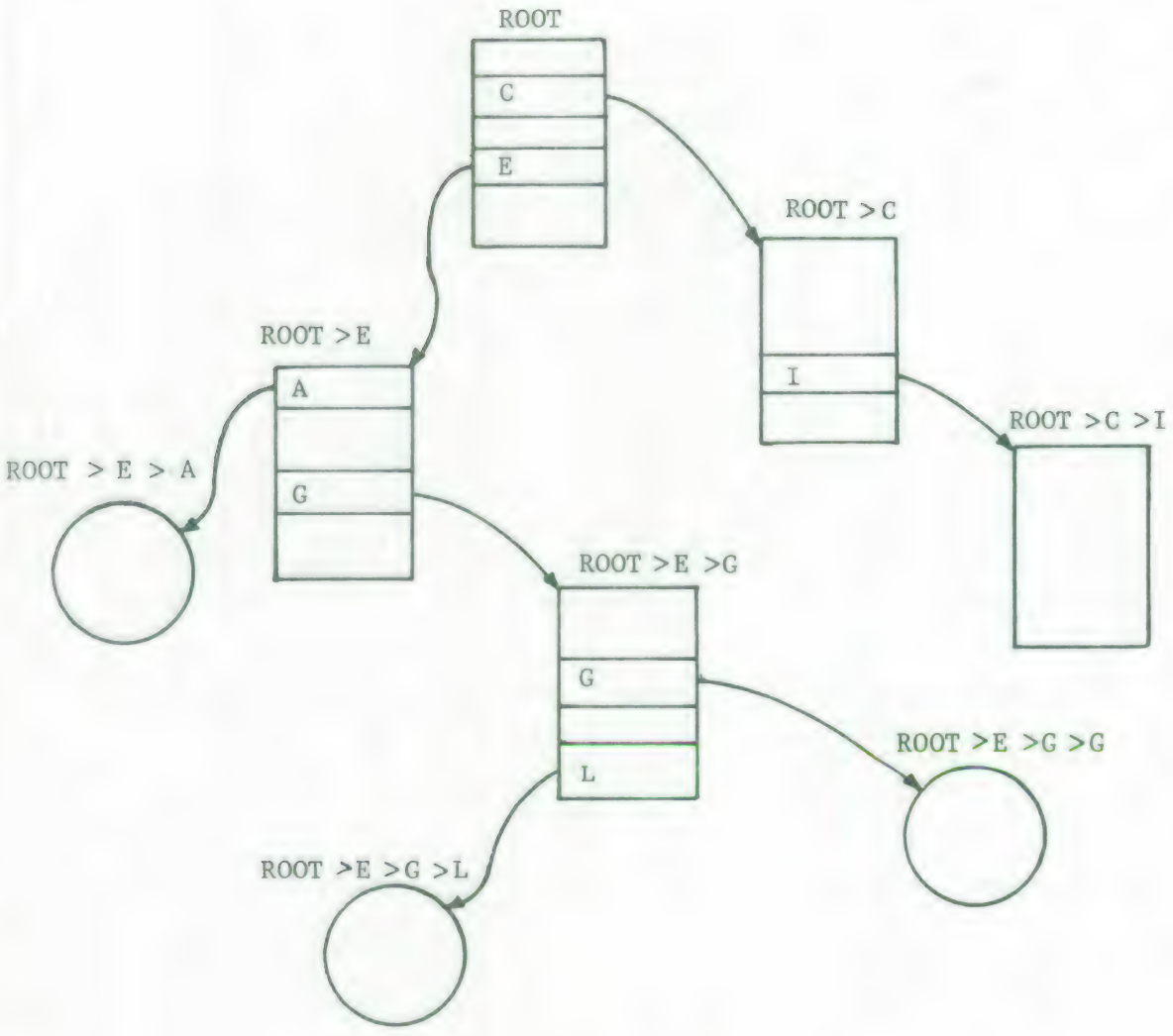
The heart of Multics is a segmented virtual memory used for storing all information in the system, including system data bases and tables.[28] A Memory cell has a two-component address--a segment name and an offset within the segment. The allocation of the different levels of physical storage (disk, bulk store, main memory, and cache) is automatically performed by system software and hardware, and is of no concern to the user.

A segment is the cataloguing unit of the storage system, and it is also the unit of protection.[29] The name of a segment and its other attributes are kept in a catalogue, implemented with segments of a special type, called directory, which are organized into a tree structure. The name of a segment is a list of subnames (called a pathname) that reflect the position of the segment in the tree (see Figure 5).

The basic protection mechanism is the access control list, an open ended list of user names or groups of users (project names) who are permitted to access a segment. An access control list (abbreviated ACL) is associated with every segment; it corresponds to a column of Lampson's protection matrix.[12] Permission to change a segment's ACL is controlled by the ACL of the segment's parent directory--thus, control of access permission is hierarchical.

Like many other contemporary time-sharing systems, Multics creates multiprogramming through the use of a process abstraction. (Generally speaking, a process is a program in execution on a virtual machine. A more precise definition can be found in [30].) A new process is created in response to each successful user logon, and deleted when the corresponding logout occurs. The implementation of the process abstraction, including interprocess communication and synchronization, and processor multiplexing is done by a subsystem called the traffic controller.[30] A process has an address space consisting of a set of segments. A process refers to segments in its address space by segment number, a process-local name that is an index into a system-maintained table of segment descriptors.

Each process is divided into eight rings of protection, hierarchical execution domains that are a generalization of the two execution domains (supervisor/problem or master/slave) found on most large-scale computers.[31] Each segment has a set of ring



Squares are Directory Segments

Circles are Data Segments

Arc labels are pathnames

Figure 5. Multics Directory Hierarchy

brackets defining the permitted modes of access in each ring. Thus, a segment may be read-write accessible in an inner ring, read-only in a middle ring, and inaccessible in an outer ring. The specific modes in which a process can access a segment are constrained by the ring brackets and the ACL.

KERNEL STRUCTURE

Figure 6 shows the relationship between the current Multics and a kernel-based Multics. In the current system, the supervisor does not operate in a dedicated process or address space. Instead, most of it is "distributed"--shared among all Multics processes. This structure facilitates communication between user and supervisor procedures and the simultaneous execution, by several processes, of supervisory functions. For protection, supervisory and user procedures execute in separate rings. The distributed supervisor executes primarily in ring 0, although some of it executes in ring 1. Applications and user software (including compilers and the command language interpreter) execute in ring 4. Some of the supervisor executes in dedicated (daemon) processes.

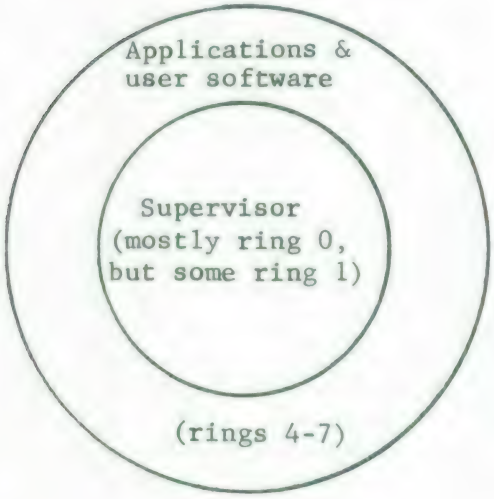
The Multics kernel can be thought of as a refinement of the current Multics supervisor. The refinement is basically a removal of functions not necessary for the implementation of the reference monitor mechanism. Removal of these nonessential functions is desirable because it leads to a minimal kernel.

Most of the Multics kernel is distributed across all processes in the same manner as the current supervisor and for the same reasons. For compatibility, supervisory functions removed from the kernel can be performed by system software that executes in the user ring (e.g., the command language interpreter) or in the non-kernel supervisor shown in Figure 6. The job of the non-kernel supervisor is to map the kernel interface into the user (ring 4) interface and to provide a "break proof" environment for the user.[32]

Protection Rings

The kernel-based Multics shown in Figure 6 uses the protection ring mechanism provided by the base hardware [31] for three distinct purposes: 1) to isolate the kernel from all non-kernel software; 2) to isolate the non-kernel supervisor from user software; and 3) to internally partition the kernel software. Each warrants some explanation.

Current Multics System



Secure, Kernel-based Multics System

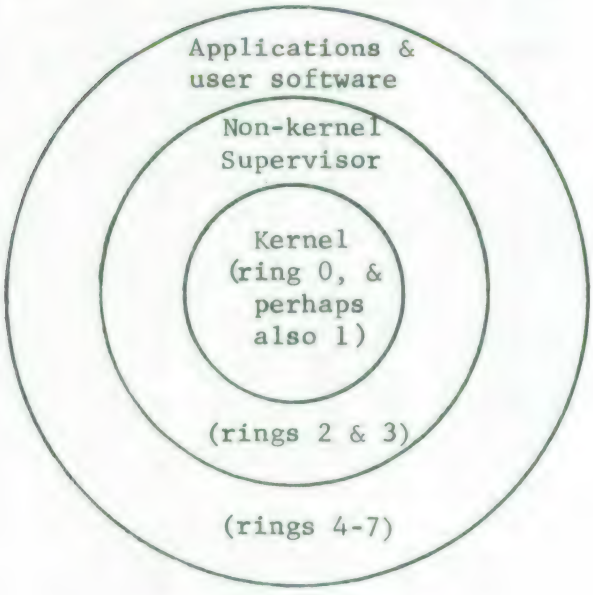


Figure 6. System Structure

Isolation of kernel software is a basic requirement of the reference monitor/security kernel concept. Isolation is achieved by having only kernel software execute in the innermost protection rings, and by using the "gate" feature of the ring mechanism to define the allowable entry points to the software-implemented kernel functions. The ring bracket component of segment descriptors, which are stored in descriptor segments, prevents non-kernel software from modifying the kernel's procedures and data base, including descriptor segments.

Although the isolation of the non-kernel supervisor from user software is not required for security, it is useful for general system reliability and utility. For example, accounting functions will be performed by the non-kernel supervisor. Isolating it prevents user software from tampering with accounting data.

Isolation of the non-kernel supervisor can be achieved if the kernel does two things: 1) limits access to its software-implemented functions to the non-kernel supervisor; and 2) allows the user of its functions to define how the ring brackets on non-kernel segments should be set. Step 1) simply involves setting the "call bracket" on kernel gates so that they can only be called from the supervisor rings, not from user rings. The supervisor can then use 2) to have the ring brackets on its segments set so that they cannot be modified by user ring software. User software will only be able to call supervisor functions and execute the unprivileged machine instructions.† Ring brackets will constrain user instruction execution to manipulating user ring segments. Thus, isolation of the non-kernel supervisor adds minimal complexity to the kernel. In determining whether or not to perform a requested function, the kernel need only consider security requirements, not the requirement for supervisor isolation.

Partitioning kernel software into two rings may aid verification. Partitioning can be done on the basis of the two types of security policy enforced by the kernel, non-discretionary and discretionary. Since non-discretionary security (i.e., based on DoD security levels) is more important than discretionary security (i.e., DoD "need-to-know"), the kernel software that provides non-discretionary controls may be subjected to a more rigorous and formal verification than the kernel software that provides

---

† It may be possible to provide user software with direct access to some kernel functions (to avoid supervisor overhead) while still maintaining supervisor isolation. Since supervisor isolation is not a security issue, the determination of the kernel functions to which user software can have direct access can be made on an ad hoc basis.

discretionary controls. Separation of these two classes of kernel software by a protection ring prevents the "less verified" discretionary control kernel software from compromising the non-discretionary controls.

The structure we have then is a ring 0 kernel that enforces non-discretionary security and a ring 1 kernel that builds on the ring 0 kernel and adds discretionary controls. The separation is hidden from non-kernel software--all it sees is a kernel that provides non-discretionary and discretionary security. It is this latter kernel that is our interest in this report.

The determination on how thorough and rigorous the kernel verification must be is a policy, not a technical, issue. Even if all kernel software is verified to the same degree, however, an internal partitioning of it into separate execution domains may aid the verification. With the current Multics hardware base, it is feasible to allocate only one or two of the eight protection rings to the kernel. A machine that supported the more general form of non-hierarchical execution domains would allow a more complete internal partitioning of the kernel.

Kernel Subsystems

The kernel consists of six subsystems divided into two groups: primary and secondary. The three primary subsystems are required to support each user process. Thus, compatibility and efficiency are important issues in the design of these subsystems. The three secondary subsystems do not provide any functions to user processes. Hence, compatibility and efficiency are not as important for the secondary systems. In the remainder of this volume we will concentrate on the primary subsystems, and limit our treatment of the secondary subsystems to a general overview.

PRIMARY KERNEL SUBSYSTEMS

The three primary kernel subsystems are the storage system, process management, and external I/O. In this subsection we outline the functions and security requirements associated with each subsystem, discuss design alternatives, and indicate the reasons for selecting among the alternatives. A more detailed description of the subsystem functions is given in Section IV.

Each subsystem creates abstractions and provides operations for manipulating them. To give the reader a feel for the security correctness of the design, we will explain the correspondence of

these abstractions and operations to the elements of the security model and the suggested set of model rules given in Section II. The reader should remember that the formal verification of the kernel involves analyzing the information flows that the kernel operations permit with respect to the model axioms; the model rules are not specifically used. Thus, exact correspondence of the kernel functions to model rules is not necessary.

Storage System

The kernel storage system must support the current Multics segmented virtual memory. Given this requirement and the nature of the base hardware (extensive support for a segmented virtual memory), the kernel should clearly provide a segmented virtual memory. There are, however, two alternatives for the structure of this memory: 1) a tree-structured directory hierarchy; or 2) a flat file system. In a flat file system, there is (conceptually) a single, kernel-maintained segment catalogue, and all segments are identified by a single, fixed size unique name. Proponents of the flat system approach argue that it requires less mechanism to implement a flat system than a directory hierarchy. Thus, the flat approach should lead to a smaller, simpler, and more easily verified kernel. Another advantage of a flat file system is that it is more general than a hierarchical system. In the context of a kernel to support the existing Multics system, however, this generality is of no value.

A basic problem with the flat system approach is discretionary security. It is not known how to provide a flat file system with discretionary access controls and retain compatability with the current Multics. The current Multics access control list (ACL) mechanism does, however, satisfy discretionary security requirements. Since ACL's are integrated into the directory hierarchy, the kernel design is based on alternative 1)--the kernel storage system provides a segmented virtual memory organized in a tree-structured directory hierarchy.

In terms of the two ring kernel structure previously discussed, it may be reasonable for the ring 0 kernel to provide a flat file system with non-discretionary security controls. On top of the flat system, the ring 1 kernel would then implement a directory hierarchy with ACL's to provide discretionary control. A detailed discussion of the issues involved in this structure is beyond the scope of this report.

Given the decision to support a directory hierarchy within the kernel, we must next decide just how much of the current storage

system the kernel should support. One alternative is to perform the complete storage system implementation within the kernel. Another alternative is to identify a subset of the storage system to be implemented by the kernel, and have the non-kernel supervisor complete the implementation.

A way of identifying a subset is in terms of segment attributes. The current system maintains a number of segment attributes that are not security-related, for example, multiple segment names and segment author (the person who created the segment). One can envision a system where only one segment name is maintained by the kernel (in the segment's directory), and any additional segment names and the author attribute are maintained by the supervisor in its own data base. Although this approach has potential inefficiencies, we employ it in the design because it leads to a smaller and simpler kernel. A complete discussion of the division of segment attributes and the way in which the supervisor can create a compatible interface is given in Volume II of this report.

Another design issue in the storage system area is the management and accounting of storage resources. The kernel must manage secondary storage because the ability to store information in the virtual memory is dependent on storage availability. Without the necessary controls, storage availability can be manipulated to create an information channel that violates the *-property. The kernel must supply accounting data to the non-kernel supervisor because it is relatively easy for the kernel to collect the data (since it creates the basic storage object abstractions), and it would be difficult, if not impossible, for the supervisor to infer the data.

The current Multics provides a single quota mechanism for: 1) the limitation of virtual (user observable) resources; 2) the management of real resources (particularly disk storage); and 3) the computation of charges for resources used. Although only a few minor changes are required to make the input/output behavior of the current quota mechanism secure, it is generally considered a problem because of the complexity of its specification and implementation. Since a better alternative has not been found, however, the current mechanism, with a few simplifications that are described in Volume II of this report, has been retained.

A fourth design issue is the nature of kernel support for message segments. The current Multics includes a ring 1 subsystem that creates a message segment abstraction (built on data segments) and provides operations for entering messages, reading messages, and deleting messages in message segments. Most users employ a message segment as a mailbox for receiving mail (messages) from other users.

Message segments are also used as a request queue for system services.

In a multilevel system, it is desirable that a low level process be able to enter a message into a high level message segment.† Given this facility, there are two design alternatives: 1) the message retains the access level of the entering process (multilevel message segments); or 2) the message is upgraded to the access level of the message segment (single level message segments). Since the storage of messages requires physical resources, the secure support of alternative 1) requires a storage quota for each possible message level that may be stored in a message segment. That is, it requires a quota mechanism over and above the current mechanism, which controls storage allocation at a granularity no finer than a segment. To avoid the complexity associated with an additional quota mechanism, the kernel design provides alternative 2)--single-level message segments. The non-kernel supervisor can, however, create a multilevel message segment abstraction by making a logical association among several single level message segments. For example, if a user requests a mailbox capable of holding unclassified, confidential, and secret messages, the supervisor can create three single level message segments, all immediately inferior to the same unclassified directory. Any process between unclassified and secret can determine the existence of these three segments by observing the directory, and enter a message in the appropriate segment.

The final storage system design issue that will be discussed here involves kernel support for demountable logical volumes. A recent enhancement to the Multics Storage System are features that allow: 1) a set of segments to be grouped on the same (logical) disk volume; and 2) the demounting of the disk volumes supporting the virtual memory. (A logical volume is one or more physical volumes). The current design requires a process to request the mounting of a volume before it accesses a segment in it, even if the volume has been mounted at the request of some other process.

Whether or not a mount request can be satisfied depends in part on the availability of a physical resource, disk drives. To prevent the mounting and demounting of volumes from creating a storage channel in violation of the *-property, a kernel must partition disk drives on a per access level basis. Physical disk drives and demountable volumes must have access levels, and volumes can only be mounted on drives of the same level, at the request of a process

†In fact, this ability is required by the kernel design for external I/O.

at the same level. With the requirement for each accessing process to request a mount, then only segments at a single level can be stored on a demountable volume, because a process can only write into a segment at its own level. (This restriction does not apply to permanently mounted volumes.)

Since single level demountable volumes will probably not be acceptable to most installations, the kernel design does not follow the approach outlined above. Instead, each mount request is transmitted to the System Security Officer via the message segment facility. The SSO determines whether or not to honor each mount request, and directs the system operator to act accordingly. Since this design does not virtualize disk drives on a per access level basis, a potential *-property violation is created. The design is acceptable, however, because the SSO is directly involved, and he can prevent the channel from being exploited.

Storage System Functions

The basic objects created by the kernel storage system are data and directory segments, organized as a directory hierarchy. Each segment (except the Root directory) has a parent directory. A directory is an array of branches, one branch for each immediately inferior segment. The branch contains the inferior segment's attributes including its name (with respect to its parent directory; i.e., the last component of its pathname), access level, and ACL. A segment's ACL corresponds to a column of the model's discretionary access control matrix.

Functions are provided for creating and deleting segments, adding elements to and removing elements from a segment's ACL, and getting and releasing access to segments. These functions correspond directly to the model rules described in Section II, as shown in Table I. Since the correspondence between model rules and kernel functions is based on the similar information flows associated with each, the security constraints on the kernel functions are similar to the constraints on the corresponding rules.

Functions are also provided for accessing segment contents and for mounting and demounting disk volumes. Mount and Demount observe and modify volume state information. These functions correspond to the implicit (non-transition) model rules, which permit a subject to access the objects in its current access set in accordance with the axioms.

Table I

Storage System Correspondence

Multics Kernel	Model
Transition Functions	
Create_segment	create object
Delete_segment	delete object
Add_ACL_element	give and
Remove_ACL_element	rescind access
Initiate	get access
Terminate	release access
Revoke	
Segment Accessing Functions	
Seg_attributes	observe (directory) object
Seg_side_effect_attributes	
Move_quota	observe and modify (directory) object
Release_page	
Enter_msg	modify (data) object
Write (generic hardware functions)	observe and modify (data) object
Read (generic hardware functions)	observe (data) object
Volume Control Function	
Mount	observe and
Demount	modify object

Process Management

The process management kernel subsystem creates the process abstraction and provides functions for the secure manipulation of processes. The basic design issues in this area center on kernel support for user authentication. Associated with each Multics process is a principal identifier, consisting of a user name, a project name, and a tag. (The tag component is only partially supported by the current implementation, and is not important for DoD security.) The intent is that a process's principal identifier identifies the person on whose behalf the process is executing. Whenever a process attempts to access a segment, its principal identifier is compared with those appearing in the segment's ACL. If no match is found, access is denied.

The procedure of establishing the identity of an individual who wishes to login to the system is called user authentication. In Multics, user authentication is done by the Answering Service subsystem. If the Answering Service determines that an individual requesting to login is an authorized user (with a mechanism commonly employed in time sharing systems--passwords), it creates an interactive process with an appropriate principal identifier for the user. Multics also supports batch operation with what is called absentee processes. While one absentee process may request the creation of another, the original request for an absentee process must come from an interactive process. Thus, the basis of all process principal identifiers is the interactive user authentication procedure using passwords.

There are two alternatives for the placement of the Answering Service in a kernel-based Multics: 1) design the Answering Service into the kernel; or 2) provide kernel functions that allow the Answering Service to operate in the non-kernel supervisor. The basic argument for alternative 1) is that the utility of the ACL mechanism depends on kernel-performed user authentication. There is, however, a two-part argument against alternative 1). First, since ACL's are inherently susceptible to a Trojan Horse attack, nothing is lost by performing user authentication in the supervisor. In particular, a user's intention for ACL-based protection can be subverted by any malicious or erroneous software executing in his process in a user ring, but the supervisor can prevent user-ring software from subverting supervisor-provided user authentication. Second, performing user authentication in the kernel adds significant complexity to the kernel because it requires that the complete login dialogue, including terminal/communications I/O, be implemented entirely by kernel software. Communications I/O software, in particular, can be very complex.

Based on the arguments against alternative 1), the design follows alternative 2)--the kernel provides functions that support a supervisor implementation of the Answering Service. It is important to note that this decision implies no degradation of the effectiveness of non-discretionary security. Authentication with respect to access level (the basis of non-discretionary security) is assured by controlled entry to terminal rooms and the current access level attribute of terminals (a type of object). The kernel functions that support the Answering Service are Create_process and Set_principal_identifier. The gross operation of these functions should be obvious from their names; in Section IV we will describe how the Answering Service uses them to support user login.

The process management functions and the correspondence to model rules are shown in Table II. With respect to security, processes are non-hierarchically structured objects.† The Create_process and Delete_process functions correspond to the similarly named model rules; the security constraints described in Section II apply to these kernel functions as well. The other functions, Set_principal_identifier and the interprocess communication functions, correspond to the implicit (non-transition) model rules, and simply access information in a manner consistent with the simple security condition and *-property.

External I/O

External I/O is the transmission of information between the internal (secure computer system) and external (secure people/paper system) environments. A basic design philosophy is that external I/O should be performed by non-kernel software, subject to security-related access controls by the kernel. Engineering considerations dictate a distinction between two types of external I/O: communications I/O, primarily to terminals; and peripheral I/O--to card readers, printers, and tape drives.

Communications I/O is supported by a secure front end processor (SFEP) with distinct kernel and non-kernel software. The two kernels (Multics mainframe and SFEP) provide a mechanism that allows communication between the non-kernel software in the two machines. It is expected that the physical connection of a secure Multics to a computer network will be via the SFEP, and that the network control program will execute in the SFEP. Thus, a Multics user process will use the SFEP communication mechanism as the first step in establishing a connection with another network host.

†Specifically, the identity of a process's parent process is never considered in any of the process management kernel functions.

Table II

Process Management Correspondence

Multics Kernel	Model
State Control	
Create_process	create object
Delete_process	delete object
Set_principal_identifier	modify object
Interprocess Communication	
Wakeup	modify object
Block, Interrogate	observe object

Peripheral I/O devices are controlled by Multics non-kernel software. Since control requires both observation and modification of the device state, a process can only control devices at its level. Like most current architectures, the Multics hardware gives software direct, physical access to all I/O devices or to no devices. Thus, to permit a process to control devices at its level and prevent it from accessing all other devices, the kernel must support interpretive access to I/O devices. With an appropriate hardware architecture [34, 35], kernel support of interpretive access would not be required. (The SFEP has such an architecture.[33])

In the current Multics, line printers and card readers are controlled by supervisor daemon processes, one daemon for each device. Printer daemons are managed by another daemon, the I/O coordinator (IOC). A request for line printing ("dprint" request) from a user process is sent to the IOC, which in turn forwards it to an appropriate daemon. The IOC also interprets the dprint "delete" option. If this option is used, a segment is deleted after it is printed. Card decks (with user identification) are read by the card reader daemon into segments temporarily stored in a system directory. When a user logs on, he can move any segments containing images of his card decks to his own directories.

The main design issue in the area of external I/O involves the ways in which the kernel supports the I/O structure outlined above. As discussed in [36], in an environment where I/O devices are controlled by unverified software, there is a fundamental difference between input-only devices (for example, card readers) and output-only devices (printers). The security policy permits low level information to be printed on a high level printer. For example, a printer daemon with an access level of secret can print confidential, as well as secret, segments on a printer with a current level of secret. All output must be tentatively classified at the level of the device (secret in our example), but manual procedures external to the computer system can be used to downgrade the output to its proper level. If a card reader, however, is at the secret level, it is not reasonable to read anything but secret card decks with it. Reading decks of higher levels is a policy violation, and reading decks at lower levels leads to overclassification of the decks once they have been read into the system. This overclassification can only be corrected by an SSO operation. Given the need for an SSO operation, it is better to reconfigure the card reader before the deck is read than to downgrade the deck after it has been read, because the former approach leaves less room for error. Consequently, card reader I/O can be implemented in a secure Multics without

specialized kernel support. The main requirements are the ability for the SSO to reconfigure the current access level of card readers and the ability for the operator to login and logout card reader daemons at the appropriate levels. Generalized kernel functions to support these operations are available.

It remains for us to consider printer I/O. Since printer daemons will be implemented with non-kernel software, this task reduces to an analysis of the IOC. One design approach is to implement the IOC within the kernel. We would like to avoid this approach because it adds size and complexity to the kernel. At a minimum, the IOC must deal with multiple priority queues, support interpret the delete option. Another approach is not to provide any support for the IOC. We will see that with this approach it is not always possible to fully support the dprint delete option. For this reason we have settled on a third approach--provide enough kernel support for a non-kernel IOC to implement the dprint delete option.

In developing the design, we must consider the ways in which an installation can structure its printers and printer daemons. The two extremes are: 1) all printers are always system high (and thus operated by system high daemons); and 2) printers are reconfigured to all levels at which segments are printed, as required by the load. To reconfigure a printer, an operator logs out the daemon that is currently controlling the printer, then an SSO changes the level of the printer (and perhaps the printer forms are also changed), and finally an operator logs in a new printer daemon at the new level of the printer. An intermediate position between these two extremes is: 3) associating a range of access levels with each printer with the printer being controlled by a daemon at the high end of the range. For example, a top secret printer (controlled by a top secret daemon) could print top secret and secret segments, but not confidential segments. Ranges can be chosen so that only occasional printer reconfiguration is required, or so that reconfiguration is only necessary when a printer breaks down.

If all printers are always system high (case 1 above), then it is a simple matter for the non-kernel supervisor to provide the IOC. The IOC, like the printer daemons, is a system high process, and user print requests are sent "up" to it via a system high message segment. The IOC in turn forwards the requests to printer daemons as appropriate. Without any kernel support, there are two incompatibilities with this approach. First, the delete option cannot be supported because the *-property prevents the system high IOC or printer daemon from deleting a segment in any user directory below system high. And second, user processes cannot receive any status information about their print requests.

The first problem can be solved if a segment to be deleted after printing is copied up into a system high IOC directory at the time the print request is made. Then, after it is printed the IOC can delete it. The copy up can be implemented in at least two ways: 1) after the IOC receives the user request (via the message segment mechanism) it copies the user segment into its system high directory, and then, at a later time, the user process deletes the segment from its own directory; or 2) to avoid the synchronization problem introduced by the first method--the user process cannot be told when the IOC has finished its copy--the kernel can provide a function to do the copy. Strict adherence to the *-property prevents the user from being told whether or not the copy was successful. (It must fail if there is no room in the IOC directory.) Thus, in either case, the segment can be deleted without being printed. This anomaly is not a serious problem because most segments that are printed with the delete option can be easily recreated (e.g., the source listing from a compilation) or retrieved from backup storage.

The second problem, lack of status information for the requesting user, can be addressed by procedural downgrading. The printer daemons can record the segments that they print in a separate, system-high segment for each level of segment printed. Periodically, the SSO can inspect and downgrade these log segments to the appropriate level, and the supervisor can use them for reporting status information back to the user. Naturally, this procedure opens up a channel that can be exploited by a Trojan Horse attack. Each installation can use its own threat assessment to determine if this procedure's utility is worth the risk.

If printer daemons at different levels are introduced (cases 2 and 3 above), the situation becomes more complex, but still manageable. Since effective IOC management of a printer daemon requires two-way communication, an IOC process must be at the same level as the daemons it manages, and there must be a separate IOC process for each level at which printer daemons operate. If printer daemons are logged in and out as printers are reconfigured (case 2), then it is probably appropriate to log the controlling IOC's in and out also. The mechanism used for requesting a dprint, a message segment, must always be available to a user process. There must be a separate message segment for each level at which IOC's operate.

If the IOC that will manage a user's print request is not always logged in, then the first technique described above for copying a segment into an IOC directory (so that it can be deleted after dprinting) does not work, because the user process may not know when it is safe to delete the segment from its own directory.

Thus, to allow a non-kernel IOC to support the dprint delete option, the kernel must provide a function for copying a segment into a higher level directory.

To conclude our analysis of IOC requirements, we have shown that the IOC can be implemented with non-kernel software if the kernel provides a single function to support the dprint delete option. The only incompatibility is that procedural downgrading by the SSO is necessary for users to receive status information on their dprint requests. Decisions on how to structure printers, printer daemons, and IOC's can be made by each installation, and are independent of the kernel design.

External I/O Functions

As shown in Table III, the external I/O subsystem provides functions in 3 classes: 1) functions that allow non-kernel software to control peripheral I/O devices; 2) a function to support the dprint delete option; and 3) a function for communication with the SFEP. The Get-device and Release-device functions in class 1) correspond to the similarly named model rules. Devices are the only object supported by this subsystem, and they are static. During kernel operation, devices are neither created nor deleted, and all processes always have discretionary permission to all devices. (The level of a device can be changes by a trusted subject operation.) All other functions simply access information as allowed by the axioms.

SECONDARY KERNEL SUBSYSTEMS

The secondary kernel subsystems are reconfiguration, trusted subjects, and initialization. None of the functions provided by these subsystems are available to users or user processes.

The reconfiguration subsystem allows hardware modules to be added to and removed from the hardware base. Reconfiguration is a kernel subsystem because the hardware "belongs" to the kernel--the kernel uses the hardware to create the objects available at the kernel interface. Since reconfiguration operations observe and modify the configuration state information, the kernel can only provide these operations to processes with an access level equal to the level of this information. (The actual level is arbitrary. Malicious invocation of these functions by a process at the correct level cannot lead to a security compromise, because the only information visible at the kernel interface that these functions

Table III

External I/O Correspondence

Multics Kernel	Model
Peripheral I/O	
Get_device	get access
Release_device	release access
Write_device	modify object
Read_device	observe object
IOC-support	
Copy_segment	modify object
SFEP Communication	
Send	modify object

observe and modify is the reconfiguration state information). For system reliability, the non-kernel supervisor will allow only the operator's process to invoke the reconfiguration functions. All reconfiguration functions correspond to the observe object and modify object model rules.

Since the Trusted Subject subsystem functions perform the security-related binding of computer system elements to the external environment, they do not correspond to model rules. Trusted Subject functions provided by the Multics kernel are available only to the System Security Officer (SSO), an individual trusted not to use his special privileges to compromise system security. Examples of SSO operations are changing the current access level of a peripheral I/O device and downgrading a segment.

Logins at any level above unclassified can only be made at terminals that are in (physically) protected areas. Uncleared users will not have unescorted access to protected areas. To allow these terminals to be used at more than one level, the cleared user will be able to use a Trusted Subject function provided by the SFEP to tell the Multics kernel the level at which he wants his process to operate. A more detailed description of unclassified and classified login scenarios is given in the next section.

The Initialization Subsystem is responsible for putting the system into the initial secure state required by the model. Thus, there is no correspondence of initialization functions to model rules. The basic issue in this area involves the complexity of the current initialization scheme. The complexity is largely due to the ever-changing environment in which the initialization programs run. Some simplification can be achieved by moving some of the operations currently performed by initialization code to the system tape generation procedure, which runs in a standard Multics environment, [37]

SECTION IV

DESIGN DETAILS

In this section, we complete our initial description of the Multics security kernel with the presentation of some design details. The subsections describe the functions provided by the three primary kernel subsystems. A formal, top-level specification of the functions and a discussion of compatibility issues is given in Volume II of this report. Since the security constraints associated with the kernel functions have been described in the previous section, our discussion will emphasize the constraints associated with implementation decisions.

THE STORAGE SYSTEM

The storage system functions and their parameters are listed in Table IV. Each process has an address space, and segments in the address space are identified by a segment number (termed seg, or if the segment must be a directory, dir). A segment number is a per-process local name that is an (integer) index into the known segment Table (KST), a kernel-maintained data base.[28] Storage system functions allow two means of segment identification: 1) by segment number (seg), if the segment is in the process's address space; and 2) by the parent directory's segment number and the segment's entry name (dir, entry), if the parent is in the process's address space. (A segment's entry name is the last component of its pathname; it identifies the segment with respect to its parent directory.) In an experimental implementation, this approach to segment identification was shown to simplify the current ring 0 storage system.[38]

As mentioned in Section III, the kernel design retains the quota mechanism of the current Multics, with some minor modifications required by the security policy and two small simplifications. The mechanism is described in detail in Volume II of this report, and its effects are included in the formal specification. For the purposes of this section we will simply state that any storage system operation that stores data into a segment is aborted if the completion of the operation would require additional storage resources and the relevant quota account is exhausted.

Volume Control

Before a process can access a segment stored on a demountable disk volume, the process must use the Mount function to request that

Table IV

Storage System Functions

Volume Control

Mount (volume-name)

Demount (volume-name)

Address Space Control

Initiate (dir, entry, seg, ring brackets, gate flag, call limiter)

Terminate (seg)

Revoke (dir, entry)

Directory Operations

Create_segment (dir, entry, type, access_level, quota, sons-volume-id)

Delete_segment (dir, entry)

Add_ACL_element (dir, entry, acli, principal-identifier)

Remove_ACL_element (dir, entry, acli)

Accessing Segments

Read (seg, offset) returns data) generic

Write (seg, offset, data)) hardware functions

Enter_message (seg, message)

Seg_attributes (dir, entry) returns attributes

Seg_side_effect_attributes (dir, entry) returns attributes

Move_quota (dir, entry, quota)

Release_page (seg, page)

the SSO mount the volume. We assume that the non-kernel supervisor will enforce a resource control policy that prevents a process from monopolizing disk drives.

When a process has completed its use of a volume it calls the Demount function. If no other processes have the volume mounted, the SSO may free the disk drive.

Address Space Control

A process can only access a segment in its own address space. The Initiate function moves a segment into a process's address space and associates a segment number with it; the Terminate operation performs the inverse operation. The modes in which the process may access the segment with respect to non-discretionary security are evaluated at Initiate-time. Directories at higher levels cannot be initiated because there are no modify-only operations on initiated directories. A process's discretionary access rights to a segment are determined each time the segment is accessed.

Initiate parameters include ring brackets, a gate flag, and a call limiter. These parameters allow non-kernel software to set the ring brackets on non-kernel segments to any non-kernel rings. Using this feature, non-kernel software can enforce a protection policy over and above non-discretionary and discretionary security. The protection policy is constrained to operate within the limitations of security requirements and must be consistent with the ring mechanism, that is, it must be hierarchical. One obvious application is the protection of the non-kernel supervisor from non-kernel software as discussed in Section III.

To allow changes in the ring brackets of non-kernel segments to be effective immediately, the kernel provides a Revoke operation. With this function, a process can remove a segment from the address space of all processes. Subsequent attempts to access the segment will cause a fault that can be handled by the supervisor. If appropriate, the segment can be reinitiated with new ring brackets, and the faulting access restarted.

Directory Operations

Functions are provided for creating and deleting segments. The parameters of the Create_segment function identify the directory in which the new segment is to be created, an entry name for the new segment, its type (data or directory), its access level, a storage quota allocation, and a "sons-volume-id" attribute. The

sons-volume-id attribute applies only to directories; it specifies the disk volume on which all data segment sons of the directory are stored. All directories are stored on the "root" volume, the disk volume on which the root directory is stored.

The Delete_segment function has three distinct effects: 1) the segment's branch in the parent directory is cleared; 2) the segment is removed from all process address spaces; and 3) resources allocated to the segment (including storage quota) are recovered.

The Add_ACL_element and Remove _ACL_element functions allow a segment's Access Control List (ACL) to be changed. Both functions include a parameter (acli) that specifies the position in the ACL where the new element is to be added or identifies the existing element to be removed. The task of maintaining a particular ordering of ACL elements based on the "don't care" indicators in the three fields of the principal identifier is left to the non-kernel supervisor. (For details on this subject, see [29].)

Accessing Segments

A process can access segments in its address space directly (with machine instructions) or interpretively, by calling functions implemented with kernel software. Machine instructions are constrained by the access permission bits in segment descriptor words (SDWs). In constructing an SDW, the kernel factors in all security requirements. For example, the R and W bits in an SDW (enabling a process to load from and store into a segment) are both set if and only if the segment and process are at the same level and the process has read and write discretionary access permission. For data segments, the SDW ring brackets are supplied by the user at Initiate-time, but the SDW ring brackets for a directory are always the kernel's, forcing users to access directories interpretively. The non-kernel supervisor can (and must for compatibility with the current Multics) associate additional ring brackets with directories, and use them in mediating access to the kernel functions that operate on directories.

Machine instructions that access segments correspond to a generic Read function, a generic Write function, or a combination of the two. The Write function is both observation and modification because error information is returned to the user (e.g., quota overflow). To allow the supervisor to support a message segment facility, the kernel provides an Enter_message function. From the user's point of view, this function performs modification without observation--the user is not told if his message was successfully entered.

The kernel cannot permit users to access directories with machine instructions for two reasons. First, the correct operation of the kernel depends on the information stored in directories (e.g., access levels). Therefore, it must control the way the information is modified. Second, a directory can contain information at multiple levels. To limit a process to observing only that information at its level and below, the kernel must force the user to observe directories interpretively. Specifically, a directory contains two classes of information for each inferior segment: 1) attributes set by explicit directory operations; and 2) attributes set as a side effect of modifying the segment. Class 1) attributes are at the level of the directory and class 2) attributes are at the level of the segment. Functions are provided for observing the attributes in both classes.

The final two storage system functions are involved with the quota mechanism. Their operation is described in Volume II.

PROCESS MANAGEMENT

The Process Management subsystem provides functions in two categories: process state control and interprocess communication. The functions are listed in Table V.

Process State Control

The process state control functions include Create_process and Delete_process; their effects should be obvious. Although the kernel will permit any process to create new processes and delete existing processes at equal and higher levels, in practice, the supervisor will restrict the ability to create new processes to the Answering Service process, and restrict a process's ability to delete processes to deleting itself.

The Set_principal_identifier is provided so that user authentication can occur after the Answering Service has created a new process in response to a login attempt. When a cleared user turns on a terminal in a protected area, the SFEP kernel makes a trusted subject function, Set_terminal_level, available to the user. With this function, the user tells the kernel the level at which he wishes to operate. In response, the SFEP and Multics kernels set the current access level of the terminal and send messages to system-low, Answering Service processes in both machines indicating that a login attempt is in progress, and the access level of the attempt. On the Multics side, the Answering Service creates a new process at the requested level with a default principal identifier. This new

Table V

Process Management Functions

State Control

Create_process (access-level, principal-identifier, initial execution point) returns process-id

Delete_process (process-id)

Set_principal_identifier (principal-identifier)

Interprocess Communication

Wakeup (process-id, notice)

Block returns notice

Interrogate returns notice

process establishes a connection with the SFEP process that provides its terminal I/O and starts the user authentication procedure. If the user is found to be an authorized system user, the Set_principal_identifier function is invoked to change the process's principal identifier from the default to the correct value, otherwise the process deletes itself. Login attempts from unclassified terminals can be handled completely by non-kernel software.

Interprocess Communication

Interprocess communication is supported by the block/wakeup mechanism. Wakeup sends a small (72 bits in the current implementation) message to another process and signals the specified process of this state. Block either returns the messages pending for the process that executes it, or, if there are no messages pending, suspends the execution of the process until a message for it arrives and causes a ready process to start running. The Interrogate function performs the same operation as block, but if no messages are pending, it returns immediately to its caller. The task of creating event channels is left to the non-kenel supervisor.

EXTERNAL I/O

As discussed in Section III, peripheral I/O devices are attached directly to the Multics mainframe via IOM's, but communications devices are connected to Multics via a secure front-end processor (SFEP). As shown in Table VI, the kernel supports this separation with distinct sets of functions.

Communications I/O

To have communications I/O performed on its behalf, Multics non-kernel software must communicate a request to SFEP non-kernel software. Since each machine will be operating in a multilevel security mode, the communication mechanism must be provided by the two kernels, so that the appropriate security controls can be enforced.

The design provides a process-to-process communication path based on the block-wakeup mechanism. Since both kernels support this mechanism, communication is established with the Send function, which gives non-kernel software in one machine access to the Wakeup kernel function in the other machine. SFEP software will also be able to interrupt a Multics process (to transmit a quit signal). As yet, no need for Multics non-kernel software to interrupt an SFEP process has been identified.

Table VI

External I/O Functions

SFEP Communication

Send (SFEP-process-id, message)

Peripheral I/O

Get_device (device-name)

Release_device (device-name)

Write_device (device-name, data-buffer, status)

Read_device (device-name, data-buffer, status)

IOC - support

Copy_segment (dir, entry, to_dir, to_entry)

Peripheral I/O

The Multics kernel will give non-kernel software interpretive access to peripheral devices. As described in [39], security requirements dictate three major types of I/O function: 1) authentication; 2) controlled attachment; and 3) controlled operation. Authentication establishes the identity of the user or I/O medium at the device. Once authentication has been performed, the internal security controls know the security attributes of the I/O device. Attachment is the (usually software) connection of the device to some process in the computer system. Finally, controlled operation is the mechanism that enforces the allowed attachments.

The access levels of peripheral I/O devices are known to the kernel and can be changed by a security reconfiguration. It is the responsibility of operations personnel to mount I/O media on devices in a manner that is consistent with the access levels of the medium and device, and the security policy. For example, a magnetic tape that is not write protected can only be mounted on a tape drive at the same level, where as a write-protected tape can be mounted on an equal or higher level drive.

Controlled attachment is supported with the Get_device function. It allocates a device to a process; deallocation is performed by Release_device. A process will only be allocated devices at its own level. Controlled operation is provided with the Write_device and Read_device primitives. With them, a user process can construct an I/O program and direct the kernel to start it in execution on the attached device. The IOM will have sufficient protection features to prevent the user's I/O program from violating the security policy. In particular, the IOM will constrain the I/O transfer to be between the device and a per-process buffer. Device interrupts will be received by the kernel, which in turn will send a wakeup to the appropriate process.

IOC Support

The copy segment function is provided so that a user can copy a segment that is to be deleted after dprinting to an I/O coordinator directory. The copy is required if the level of the IOC is greater than the level of the segment's original directory. The function raises the level of the new copy of the segment to the level of its new directory.

The first two parameters identify the segment to be copied, the second two identify the directory into which it is to be copied. The parent of this directory must be at a level no higher

than the lowest level process intended to copy segments into the directory, so that the process can Initiate it, and thus identify the directory into which the copy is to be made with a (dir, entry) pair.

SECTION V

SUMMARY

In this paper we have presented an informal description of a security kernel for the Multics system and outlined the underlying design methodology. A formal specification of the kernel interface is given in the remaining two volumes of this report. The specification contains sufficient detail to allow decisions to be made about the ability of the kernel to support efficiently and compatibly the current Multics user interface.

In designing a kernel for a system as complex as Multics, a number of trade-offs must be considered. This paper describes the major design alternatives and the reasons for selecting among the alternatives.

The next step in the design is the identification and formal specification of the mechanisms that implement the kernel. For the most part, the mechanisms employed by the current Multics system will be retained.

REFERENCES

1. J.P. Anderson, "Computer Security Technology Planning Study," ESD-TR-73-51, Volume 1, James P. Anderson & Co., Fort Washington, Pennsylvania, October 1972 (AD 758206).

2. R.R. Schell, P.J. Downey, and G.J. Popek, "Preliminary Notes on the Design of Secure Military Computer Systems," MCI-73-1, Electronic Systems Division (AFSC), Hanscom Air Force Base, Massachusetts, January 1973.

3. L. Smith, "Architectures for Secure Computer Systems," ESD-TR-75-51, Electronic Systems Division, AFSC, Hanscom Air Force Base, Massachusetts, April 1975 (AD A009221).

4. F.J. Corbato, "PL/I as a Tool for Systems Programming," Datamation, Volume 15, Number 5, May 1969, pp. 68-76.

5. J.C. Whitmore, A. Bensoussan, P.A. Green, A.M. Kobziar, and J.A. Stern, "Design for Multics Security Enhancements," ESD-TR-74-176, Honeywell Information Systems, 1974.

6. P.A. Karger and R.R. Schell, "Multics Security Evaluation: Vulnerability Analysis," ESD-TR-74-193, Volume II, Electronic Systems Division (AFSC), Hanscom Air Force Base, Massachusetts, June 1974 (AD A001120).

7. R.P. Abbott, J.S. Chin, J.E. Donnelley, W.L. Konigsford, S. Tokubo, and D.A. Webb, "Security Analysis and Enhancements of Computer Operating Systems," NBS 76-1041, Lawrence Livermore Laboratory, Livermore, California, April 1976.

8. C.R. Attanasio, P.W. Markstein, and R.J. Phillips, "Penetrating an Operating System: A Study of VM-370 Integrity," IBM Systems Journal, Volume 15, Number 1, 1976, pp. 102-116.

9. D.E. Bell and E.L. Burke, "A Software Validation Technique for Certification: The Methodology," ESD-TR-75-54, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, April 1975 (AD A009849).

10. D.E. Bell and L.J. LaPadula, "Computer Security Model: Unified Exposition and Multics Interpretation," ESD-TR-75-306, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, March 1976 (AD A023588).

11. K.J. Biba, "Integrity Considerations for Secure Computer Systems," ESD-TR-76-372, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, April 1977 (AD A039324).

12. B.W. Lampson, "Protection," Proceedings 5th Annual Princeton Conference, Princeton, New Jersey, March 1971, pp. 437-443, (reprinted in ACM Operating Systems Review, Volume 8, Number 1, January 1974, pp. 18-24).

13. D. Branstad, "Privacy and Protection in Operating Systems," Computer, Volume 6, Number 1, January 1973, pp. 43-47.

14. B.W. Lampson, "A Note on the Confinement Problem," Communications of the ACM, Volume 16, Number 10, October 1973, pp. 613-615.

15. S.B. Lipner, ACM Operating Systems Review, Volume 9, Number 5, 1975, pp. 192-196.

16. W.L. Schiller, "The Design and Specification of a Security Kernel for the PDP-11/45," ESD-TR-75-69, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, May 1975 (AD A011712).

17. D.L. Parnas, "A Technique for Software Module Specification With Examples," Communications of the ACM, Volume 15, Number 5, May 1972, pp. 330-336.

18. W.R. Price, "Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1973.

19. J.K. Millen, "Security Kernel Validation in Practice," Communications of the ACM, Volume 19, Number 5, May 1976, pp. 243-250.

20. G.J. Popek and C.S. Kline, "A Verifiable Protection System," 1975 International Conference on Reliable Software, Los Angeles, California, April 1975, pp. 294-304.

21. L. Robinson, P.G. Neumann, K.N. Levitt, and A.R. Saxena, "On Attaining Reliable Software for a Secure Operating System," 1975 International Conference on Reliable Software, Los Angeles, California, April 1975, pp. 267-284.

22. R.W. Flyod, "Assigning Meaning to Programs," Mathematical Aspects of Computer Science, Volume 19, American Mathematics Society, Providence, Rhode Island, 1967, pp. 19-32.

23. E.W. Dijkstra, "The Structure of the 'THE' Multiprogramming System," Communications of the ACM, Volume 11, Number 5, May 1968, pp. 341-346.

24. Department of Defense, "Security Requirements for Automatic Data Processing (ADP) Systems," Department of Defense Manual 5200.28, December 1972.

25. C. Engelman, "Audit and Surveillance of Multilevel Computing Systems," ESD-TR-76-369, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, April 1977 (AD A039060).

26. F.J. Corbato, J.H. Saltzer, and C.T. Clingen, "Multics — The First Seven Years," Proceedings AFIPS 1972 SJCC, AFIPS Press, Montvale, New Jersey, pp. 571-583.

27. E.I. Organick, The Multics System: An Examination of Its Structure, MIT Press, Cambridge, Massachusetts, 1972.

28. A. Bensoussan, C.T. Clingen, and R.C. Daley, "The Multics Virtual Memory: Concepts and Design," Communications of the ACM, Volume 15, Number 5, May 1972, pp. 308-318.

29. J.H. Saltzer, "Protection and the Control of Information in Multics," Communications of the ACM, Volume 17, Number 7, July 1974, pp. 388-402.

30. J.H. Saltzer, "Traffic Control in a Multiplexed Computer System," MAC-TR-30 (Thesis), MIT Project MAC, Cambridge, Massachusetts, July 1966.

31. M.D. Schroeder and J.H. Saltzer, "A Hardware Architecture for Implementing Protection Rings," Communications of the ACM, Volume 15, Number 3, March 1972, pp. 157-170.

32. M.D. Schroeder, "Engineering a Security Kernel for Multics," ACM Operating Systems Review, Volume 9, Number 5, 1975, pp. 25-32.

33. M. Gasser, "Top Level Specification of a Security Kernel for Multics Front End Processor," ESD-TR-77-258, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, November 1977.

34. D.D. Clark, "An Input/Output Architecture for Virtual Memory Computer Systems," MAC-TR-117, MIT Project MAC, Cambridge, Massachusetts, January 1974.

35. E.L. Burke, "Secure Minicomputer Architectures," M76-224, The MITRE Corporation, Bedford, Massachusetts, October 1976.

36. J. Mogilensky, "A General Security Marking Policy for Classified Computer Input/Output Material," ESD-TR-75-89, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, September 1975 (AD A016467).

37. N. Adleman, "Effects of Producing a Multics Security Kernel," ESD-TR-76-130, Honeywell Information Systems, McLean, Virginia, October 1975.

38. R.G. Bratt, "Minimal Protected Naming Facilities for a Computer Utility," MAC-TR-156, MIT Project MAC, Cambridge, Massachusetts, September 1975.

39. E.L. Burke, "Concept of Operation for Handling I/O in a Secure Computer at the Air Force Data Services Center (AFDSC)," ESD-TR-74-113, Electronic Systems Division, AFSC, Hanscom AF Base, Massachusetts, June 1974 (AD 7805259).